Attorney Docket No.: 42P15733 *PATENT*

UNITED STATES PATENT APPLICATION

For

OPTIMIZATION OF SMI HANDLING AND INITIALIZATION

INVENTORS:
Barnes Cooper, Grant H. Kobayashi

Prepared By:
BLAKELY, SOKOLOFF, TAYLOR & ZAFMAN LLP
12400 WILSHIRE BOULEVARD
SEVENTH FLOOR
LOS ANGELES, CA 90025-1026

(408) 720-8300

Optimization of SMI Handling and Initialization

FIELD

[0001] This invention relates to the field of computer systems and, in particular, to system management mode optimizations.

BACKGROUND

[0002] Computer systems are becoming increasingly pervasive in our society including everything from small handheld electronic devices, such as personal digital data assistants and cellular phones, to application-specific electronic components, such as set-top boxes and other consumer electronics, to full mobile, desktop, and server systems. However, as systems become smaller in size and in price, the need for efficient memory allocation and system management becomes more important.

[0003] Server systems have been traditionally characterized by a significant amount of conventional memory and multiple physical processors in the same system (a multiprocessor system), wherein a physical processor refers to a single processor die or single package. The significant amount of memory available to a server system has lead to extremely inefficient allocation of memory space and wasted execution time.

[0004] Typically, in a multi-processor system, upon boot every processor arbitrates for wake-up, which may include allocating memory and relocating the processor's base address (SMBase). In the process of initializing each processor, a system management interrupt (SMI) is generated, which is handled with a default SMI handler for each processor. Usually, the processors arbitrate with a race to the flag scheme, wherein the first processor to begin handling the SMI is able to begin initialization. Initialization typically includes allocation of a separate and distinct 4kB aligned memory space for each processor, which forces one to allocate more memory than needed for system management.

[0005] Furthermore, when a system management interrupt (SMI) occurs, whether during boot or regular operation, each processor in a multiprocessor system runs a separate and distinct SMI handler to service/handle the SMI. There are two types of SMIs. The first type is an asynchronous interrupt that may be generated by the system hardware, such as when a battery is low. An asynchronous interrupt may be handled separately by any processor, since knowledge of another processor's save state area is not needed to service the request. The second type of interrupt, a synchronous SMI, that is software generated, should be handled by every processor. Typically, a software generated SMI occurs when the operating system (OS) wants a processor to enter system management mode (SMM). SMM is an environment for executing software routines/handlers that does not interfere with the OS or application programs.

[0006] In current multiprocessor systems, each processor enters SMM and then one-by-one executes a distinct SMI handler to check their registers to find out which processor generated the SMI. This requires a separate SMI handler be executed for each processor, which introduces resource contention issues, thus making updates to the SMI handler code difficult.

[0007] However, these inefficient methods of initialization and handling are not limited to multiprocessor server systems, but exist in other systems, such as mobile multiprocessor systems. Hyper-Threading Technology (HT) is a technology from Intel® Corporation of Santa Clara, California that enables execution of threads in parallel using a signal physical processor. HT incorporates two logical processors on one physical processor (the same die). A logical processor is an independent processor visible to the operating system (OS), capable of executing code and maintaining a unique architectural state from other processor in a system. HT is achieved by having multiple architectural states that share one set of execution resources.

[0008] Therefore, HT enables one to implement a multi(logical)processor system in a mobile platform. As shown above, inefficient memory allocation, processor initialization, and SMI handling exist in traditional multiprocessor systems, such as server systems. Furthermore, as multiprocessor systems begin to infiltrate the mobile realm, where resources such as memory are limited, the need for optimizations of the aforementioned inefficiencies becomes even more important.

BRIEF DESCRIPTION OF THE DRAWINGS

[0009] The present invention is illustrated by way of example and not intended to be limited by the figures of the accompanying drawings.

[0010] Figure 1 illustrates a block diagram of a device with multiple processors that share execution resources, caches, and memory.

[0011] Figure 2 illustrates a block diagram of a system with multiple processors.

[0012] Figure 3 illustrates a block diagram of a system with a physical processor having multiple logical processors.

[0013] Figure 4 illustrates a flow diagram of a first processor waking a second processor.

[0014] Figure 5 depicts a flow diagram of an illustrative embodiment for waking a second processor with a first processor.

[0015] Figure 6 illustrates a flow diagram of handling an SMI on a first and second processor with the same SMI handler.

[0016] Figure 7 depicts a flow diagram of an illustrative embodiment for handling an SMI on a first and second processor with one SMI handler.

[0017] Figure 8 illustrates a flow diagram for executing the same system management interrupt code on a first and second processor if the generated SMI is a software SMI.

[0018] Figure 9 depicts a flow diagram of an illustrative embodiment of a first and second processor executing the same SMI handler, if the SMI being handled was software generated.

DETAILED DESCRIPTION

[0019] In the following description, numerous specific details are set forth such as examples of specific memory addresses, memory sizes, and component configurations in order to provide a thorough understanding of the present invention. It will be apparent, however, to one skilled in the art that these specific details need not be employed to practice the present invention. In other instances, well known components or methods, such as routine boot-up blocks (e.g. power on self-test (POST)), specific system management mode (SMM) implementation, and specific system management interrupt handler code have not been described in detail in order to avoid unnecessarily obscuring the present invention.

[0020] The method and apparatus described herein are for optimization of memory allocation when waking a processor and optimization of system management interrupt (SMI) handling in multiprocessor systems. The method of waking a processor may occur as a result of any number of normal operations. For example, when the computer is powered on or reset, one may wake an inactive/sleeping processor. Furthermore, when a system is returning from a low power state, such as sleep, standby, suspend, hibernation, wait-for-SIPI, sleep, deep sleep, reset, or any other mode where the second processor does not respond to interrupts one may wake an inactive processor.

[0021] It is readily apparent to one skilled in the art, that the method disclosed for waking a second processor may be applicable to any level computer system (personal digital assistants, mobile platforms, desktop platforms, and server platforms), as well as any number of processors. For example, a multiprocessor system with four or more processor may use this method to wake an inactive processor with an active processor.

[0022] **Figures 1-3** depict illustrative examples of some hardware that may embody the methods described herein. The methods described may be used in any multiprocessor system; therefore, the methods will only be described in reference to **Figure 3** as not to obscure the invention with unnecessary detail.

[0023] **Figure 1** illustrates a block diagram of a device 105 with multiple logical processors. A physical processor refers to a physical processor die or a single package. A logical processor is an independent processor visible to the operating system (OS), capable of executing code and maintaining a unique architectural state from other processor in a system. Hyper-Threading Technology (HT) is a technology from Intel® Corporation of Santa Clara, California that enables execution of threads in parallel using a signal physical processor. HT includes two logical processors on one physical processor and is achieved by duplicating the architectural state, with each architecture state sharing one set of processor execution resources.

[0024] Device 105 may include a first processor 120 and a second processor 125. Device 105 may be a physical processor. Device 105 may also be an embedded system or other device having at least two processors. Processors 120 and 125 may be logical processors. For example, processor 105 may include architecture state registers 130 and 135 that each holds a unique architecture state. It is readily apparent that device 105 may include more than two logical processors that each have an architecture state register associated with it to hold a separate architecture state. The two processors 120 and 125 share the same execution resources 140, caches 145, bus 150, and memory 155.

[0025] Device 105 may also include controller 110. Controller 110 may be an Advanced Programmable Interrupt Controller (APIC) controller. Controller 110 may be used to generate a system management interrupt (SMI). Controller 110 may also be used to communicate on an APIC bus, which is not depicted, coupling the first processor 120 and the second processor 125 together.

[0026] Device 105 may also include memory 155. Memory may be any style of storage, wherein data may be stored. For example, memory 155 may be registers to store information. Memory 155 may also be another level of cache 145. Memory 155 may also be a form of system memory placed on device 105.

[0027] Memory 155 has at least a first memory location 160 and a second memory location 165. First memory location 160 may contain default system management handler code. First memory location 160 may also be 1k aligned. Second memory location 165 may be another non-1k aligned address. First memory location 160 and second memory location 165 may also be base addresses for a first processor 120 and second processor 125 respectively. Second memory location 165 may also be used as temporary storage space, when relocating first processor 120's base address. The first and second memory locations will be discussed in more detail in reference to the methods described in **Figures 4-9**.

[0028] Turning to **Figure 2**, an illustrative example of a system with multiple processors is depicted. The system may include first processor 205 and second processor 210. Processor 205 and 210 may be physical processors, wherein each processor is in its own package. The system may also include system bus 215 to couple processors 205 and 210 to controller hub 220. Controller hub 220 may also be coupled to memory 230 by a bus 225.

[0029] **Figure 3** illustrates an example of a system with multiple processors. Processor 305 may include processor 310 and processor 315, which share execution resources 330, cache 335, and system bus, 340. Architecture state registers 320 and 325 hold the unique architecture state of processors 320 and 325 respectively. System bus 340 couples processor 305 to controller hub

345. Controller hub 345 may be coupled to system memory by a second bus 350. System memory may have multiple memory locations, such as first memory location 365 and second memory location 370.

[0030] **Figure 3's** illustrative system will be used to describe the methods depicted in **Figure's 4-9**. Although **Figure 3** is being referred to in describing the methods of efficient memory allocation and system management interrupt (SMI) handling, it is readily apparent that the hardware in **Figure 1, Figure 2,** and other hardware configurations not depicted may implement the methods describe herein.

[0031] **Figure 4** illustrates a high-level flow diagram of a first processor waking a second processor. The first and second processors may be logical processors located on one processor die, separate processors located on separate packages as shown in **Figure 2,** or processors in other multiprocessor configurations. Initially, when a multiprocessor system comes out of a low power state, such as sleep, standby, suspend, hibernation, wait-for-SIPI, sleep, deep sleep, reset, or any other mode where the second processor does not respond to interrupts a first processor should initialize to an active state, while a second processor should initialize to an inactive state. An active state may include executing code or responding to interrupts. In the alternative, an inactive state may include not responding to interrupts. An inactive state may also include not executing code.

[0032] In block 405, a first system management interrupt (SMI) is received. Often a SMI is generated to request a service from a processor. Once an SMI is received, a processor enters system management mode (SMM) to service the request by running SMI handler code and routines in conventional memory, unless the processor is inactive and not responding to interrupts.

[0033] As an illustrative example, the first SMI in block 405 may be generated by a controller hub, such as controller hub 345 depicted in **Figure 3**. As another example, the first SMI in block 405 may be generated by an APIC, such as controller 110 in **Figure 1,** or by a controller located separately in the system (not depicted). As yet another example, the first SMI in block 405 may be generated by changing the logic level of a pin on a processor, such as processor 305 or a controller hub, such as controller 345.

[0034] The first SMI in block 405 may be a service request to initialize SMM, to allocate address space for system management, and/or to relocate a processors base address (SMBase). The SMBase may be the address where the system management portion of memory begins. The SMBase may also be the address where the system management portion of memory is referenced from. For example, the SMBase may be a value of 0x30000. The SMI handler for that portion of memory might be referenced to the SMBase by an offset. For example, the SMI handler may be offset from the SMBase by 0x8000 (SMBase + 0x8000), which would put the SMI handler at 0x38000 in the example.

[0035]	When the SMI is generated in a multiprocessor system both a first processor and a second processor should receive/latch the SMI. However, the second processor may not enter SMM and handle the SMI at this time, since it may be in an inactive state (not responding to interrupts). In contrast, in block 410, the first processor may be active and may handle the first SMI received in block 405. When the first SMI is received by the first processor, the first processor may enter system management mode (SMM) to service/handle the SMI.

[0036]	As shown in **Figure 5**, the first processor may handle the SMI by initializing SMM in block 505 and executing a default SMI handler in block 510. The default SMI handler may be default code located at a default memory location, such as first memory location 365. The first memory location may be 1k, 4k, or any other aligned memory range.

[0037]	Returning to **Figure 4**, a wake-up signal is generated by the first processor in block 415. The wake-up signal in block 415 may be any signal that is sent to wake the second processor from an inactive state. Since the second processor may not be responding to interrupts or executing code in an inactive state, the wake-up signal may be a bus signal that the second processor, in an inactive state, waits for to begin the waking process. In one embodiment, the wake-up signal may be a startup inter processor interrupt

(SIPI) message transmitted on a bus (not depicted), such as an APIC bus, that couples the first and second processors.

[0038] As depicted in **Figure 5** in block 515, the wake-up signal may be a vector that is based on the location in memory of a default SMI handler. The wake-up signal may be any address in memory or any reference to an address location in memory. In one embodiment, the default SMI handler is located at first memory location 365, wherein the wake-up signal is a vector based on first memory location 365. In another embodiment, the wake-up signal is based on the first memory location 365, which is a 1k, 4k, or other aligned memory address range. The second processor may require that the wake-up signal be aligned and be in conventional system memory. In yet another embodiment, first memory location 365 may be the address location where second processor 315 begins execution.

[0039] Turning back to **Figure 4**, block 420 shows the second processor waking. Block 420 may include waking the second processor to allow it to handle the first SMI received in block 405, or running basic wake-up routines, such as a power on self test (POST).

[0040] In the next block, block 425, the first SMI received in block 405, is handled by the second processor. Although, the second processor may have been inactive when first SMI in block 405 was received, and therefore, unable to service the SMI at that time; the second processor may still have latched the

SMI when it was in an inactive state. Once awake, the second processor may handle the first SMI that was previously latched.

[0041] Moving to **Figure 5**, one embodiment of handling the first SMI with the second processor is illustrated in blocks 520 and 525. In block 520, the second processor initializes SMM. Additionally, in block 525 an instruction pointer for the second processor may be patched to a second memory address, such as second memory address 370 in **Figure 3**. Second memory address 370 may be a non-aligned address no longer confined to conventional memory. Once the second processor patches the instruction pointer to second memory address 370, it is not necessary to allocate the conventional aligned memory for executing startup code. After finishing initialization and resuming, the second processor, as shown in block 530, may resume to the patched instruction pointer, which points to second memory address 370, and begin executing its startup code.

[0042] Referring to **Figure 6**, a method of handling and SMI whether upon boot or under normal operation, is described, wherein an SMI may be handled on multiple processors using the same SMI handler. In block 605, a SMI is received. SMIs may be either hardware (asynchronous), such as a battery being low, or software, such as the OS requesting a processor to change frequency or power levels. Typically, a hardware SMI may be handled by either processor without the knowledge of the other processor's save-state area. A software SMI, in contrast, often depends upon the architectural state of the processor when the SMI was generated thus requiring access to the save state area of the processor which generated the SMI. Therefore, a software generated SMI may have all the processor's in a multiprocessor system enter SMM and execute a handler.

[0043] In block 610, a first processor executes SMI code (an SMI handler) to handle the SMI for the first processor. Handling an SMI may include executing SMI code to determine if the current processor's save-state area being examined generated the SMI. Handling the SMI may also include servicing the SMI request.

[0044] As stated above in reference to Figure 4, the first processor may have a first SMBase address. The SMI code (SMI handler) may be located at a default offset from the first SMBase address. Furthermore, the SMI code may reference a target SMBase address. The target SMBase may be the address where the SMI code targets to access that processor's system management area (range in memory).

[0045] As an illustrative example, the target SMBase may be set by default to reference a first SMBase address, which is the starting address of a first processor's system management area. Therefore, when an SMI is received and the SMI handler code is executed, the SMI handler code is able to access the first processor system management area, which may include the first processor's save-state area, by referencing off the target SMBase.

[0046] After executing the SMI code in block 610 to handle the SMI for the first processor, the same SMI code/handler may be executed to handle the SMI for second processor, as shown in block 615. In **Figure 7**, an illustrative embodiment of block 615, executing the SMI handler to handle the SMI for a second processor, is depicted in a flow diagram. In block 705, the target SMBase address of the SMI handler is changed from the first SMBase address for the first processor to a second SMBase address for the second processor.

[0047] Continuing the illustrative example from above, after executing the SMI handler code with the target SMBase targeting the first SMBase, the target SMBase may be changed to target the second SMBase address. The second SMBase address may be the starting address of the second processor's system management area (memory space). Therefore, when the SMI handler is executed using the second processor's SMBase as the target SMBase, as in block 710, the same SMI code/handler is able to handle the SMI for the second processor by referencing off the second processor's SMBase address.

[0048] **Figure 8** depicts another illustrative embodiment of executing the same SMI handler code for multiple processors, if the SMI being handled is software generated. As stated above, a hardware SMI may be handled by any processor without affecting the other processor. Therefore, in block 805, a first processor executes SMI code. In block 810, either the first processor or a second processor may check to see if the SMI is software generated. It is readily apparent that either processor may check to see if the SMI being handled is software generated in any temporal order. For example, the first processor may check to see if the SMI is software generated before block 805 and then execute SMI handler code in block 805. As another example, the second processor may check to see if a software generated SMI exists after block 805. If a software SMI does not exist, then the first processor exits SMM in block 815 and returns to normal operation.

[0049] However, if the SMI was software generated then the second processor should handle the SMI as well. In block 820, the second processor may then execute the same SMI code that first processor executed in block 805.

[0050] An illustrative example of executing the same SMI code is depicted in **Figure 9** with a flow diagram. Much like **Figures 6 and 7**, the first processor's SMBase, which may be located at first memory location 365, may be the target SMBase address. After the SMI code is executed with the first processor to handle the SMI for the first processor, the target SMBase may be changed to target the second processor's SMBase address in block 905. In block 910, the SMI code is executed using the second processor's SMBase as the target SMBase. In addition, as shown in block 915, the target SMBase may be restored to targeting the first processor's SMBase address.

[0051] It is readily apparent that any combination of the first and second processor may execute the SMI code. For example, the first processor may execute the SMI code for the first processor. Then, after the target SMBase is changed to the second processor's SMBase, the first processor may execute the SMI code again to handle the SMI for the second processor. As another example, the second processor may execute the SMI code both times. As yet another example, the first processor may execute the SMI code the first time the SMI code is executed to handle the SMI for the first processor, and the second processor may execute the SMI code the second time to handle the SMI for the second processor.

[0052] Therefore, unlike current systems these optimizations may use only one SMI to be generated to wake and begin execution of a second processor. Furthermore, unlike traditional methods, which start each processor at a different default memory address, memory may be saved by sending a wake-up signal that starts the second processor at first memory address, which may be the location of a default SMI handler. Moreover, by patching the instruction pointer when the second processor handles the first SMI, the second processor may resume at a second memory address, which may be non-aligned. Allowing second processor to resume at second memory address saves the allocation of a separate aligned memory space for each processor.

[0053] In addition, handling an SMI may be done with a unified handler, by executing the same handler code with a first processor and then with a second processor. By checking to see if the SMI being handled was software generated and exiting SMM if the SMI was not software generated, one may save a significant amount of execution time. Furthermore, this implementation allows for easier platform development since all changes required for the software SMI handler may easily be contained in wrapper code without requiring modification to the software SMI handler routines.

[0054] In the foregoing specification, the invention has been described with reference to specific exemplary embodiments thereof. It will, however, be evident that various modifications and changes may be made thereto without departing from the broader spirit and scope of the invention as set forth in the appended claims. The specification and drawings are, accordingly, to be regarded in an illustrative sense rather than a restrictive sense.